2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 979-8-3503-8711-7/24/$31.00 ©2024 IEEE | DOI: 10.1109/IPDPS57955.2024.00076

HINT: Designing Cache-Efficient MPI_Alltoall using Hybrid Memory Copy Ordering and Non-Temporal Instructions

Bharath Ramesh, Nick Contini, Nawras Alnaasan, Kaushik Kandadi Suresh, Mustafa Abduljabbar, Aamir Shafi, Hari Subramoni, Dhabaleswar K. (DK) Panda
Department of Computer Science and Engineering
The Ohio State University
Columbus, USA
{ramesh.113, contini.26, alnaasan.1, kandadisuresh.1, abduljabbar.1, shafi.16, subramoni.1, panda.2}@osu.edu

***Abstract*—Modern multi/many-core processors in HPC systems have hundreds of cores with deep memory hierarchies. HPC applications run at high core counts often experience contention between processes/threads on shared resources such as caches, leading to degraded performance. This is especially true for dense collective patterns, such as MPI_Alltoall, that have many concurrent memory transactions. The ordering of memory copies during the MPI_Alltoall operation can significantly affect performance as cache-efficient access patterns could potentially reduce cache misses. However, the correct access pattern depends on various factors, including cache associativity, cache sizes, coherence protocols, and memory layouts. This paper first identifies sources of bottlenecks in performing memory operations in an Alltoall. We propose different orderings for the memory copies in Alltoall operations and study their effectiveness for various message sizes. We overcome bandwidth bottlenecks related to repeated bus requests in the cache by proposing a hybrid memory copy scheme that combines regular temporal and non-temporal stores. Then, we implement an Alltoall algorithm that dynamically picks between memory orders based on their performance for different message sizes/number of processes. To the best of our knowledge, this is the first work that explores a combination of dynamic memory copy orders and non-temporal instructions for optimizing MPI_Alltoall operations. Our proposed solutions reduce the latency versus state-of-the-art solutions by up to 10x at the micro-benchmark level and 22.2% for the CPU time per loop in distributed Fast Fourier Transforms (FFTs) using P3DFFT.**

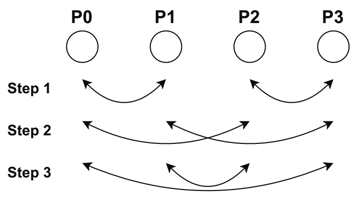
***Index Terms*—MPI, MPI_Alltoall, Hilbert curves, XPMEM, Intra-node communication, Non-temporal stores**

I. Introduction

Modern multi-core CPUs have multiple levels of memory hierarchies to cater to the needs of diverse workloads. Processor caches play an essential role in performance due to their low latency compared to other forms of memory such as DRAM, HBM, SSDs, etc., but they have limited capacity. Traditionally, caches are divided into multiple levels (in most cases, L1, L2, and L3), with L1 having the lowest latency and the latency increasing multi-fold as we traverse the memory hierarchy from L2 to L3 and from L3 to main memory. This puts the burden on applications to utilize processor caches effectively to avoid unnecessary latency hits. In a distributed setting with multiple cores, the cache behavior of communication libraries also plays an important role in the performance of applications. The Messaging Passing Interface (MPI) [1] has been a pervasive programming model in high-performance computing for distributed communication. MPI libraries often have to look at reducing cache misses by optimizing buffer accesses to be cache-efficient, especially for jobs with high core counts per node. Efficient cache usage is especially important for dense collective patterns such as MPI_Alltoall, as the significant increase in memory transactions at high core counts is likely to cause a large number of conflict misses in the cache. Due to the trend of increasing core counts on modern processors such as the AMD EPYC 9004 series [2] with 192 cores per socket, the problem of optimizing the cache usage of MPI_Alltoall within the node (intra-node) is important. Cache optimizations for MPI_Alltoall can be viewed from two standpoints – improving *spatial locality* and improving the *bandwidth of memory transactions*.

Spatial locality in caches refers to the principle that a program is likely to access memory addresses adjacent to the currently accessed memory location. Good spatial locality allows CPUs to speed up memory access by fetching multiple data elements into the cache. In the context of MPI_Alltoall, spatial locality is affected by the order of memory copies by each process. Each process in an MPI_Alltoall operation with N processes contains a contiguous send/receive buffer divided into N contiguous chunks. Globally, the set of send buffers/receive buffers can be viewed as a matrix. We define the term '*memory copy ordering*' for a process as the order of memory copies into the receive buffer matrix. For example, consider the pairwise algorithm for implementing MPI_Alltoall [3]. Figure 1(a) shows the communication pattern over time steps for a pairwise Alltoall algorithm with N = 4 processes. It consists of N - 1 remote and one local memory transfer (the process copies data to itself). The memory copy ordering is shown in Figure 1(b) in the receive buffer matrix. For process

*This research is supported in part by NSF grants #1818253, #1854828, #2007991, #2018627, #2311830, #2312927, and XRAC grant #NCR-130002.



(a) The communication pattern over time steps

Send Buffers

	P0	P1	P2	P3
P0	0	1	2	3
P1	1	0	3	2
P2	2	3	0	1
P3	3	2	1	0

Receive Buffers

	P0	P1	P2	P3
P0	0	1	2	3
P1	1	0	3	2
P2	2	3	0	1
P3	3	2	1	0

(b) Memory access for send and receive buffers visualized as a matrix. Each color represents memory operations performed by a process. Every number (say, i) in the grid denotes the ith memory copy performed in the Alltoall. 0 is for local sends/receives, whereas 1, 2, and 3 are remote sends/receives and map to the step numbers in Figure 1(a).

Figure 1: The pairwise-exchange MPI_Alltoall algorithm with *N=4*.

P0 (shown in pink), the memory copy ordering is vertical across the column in the receive buffer matrix (and hence horizontal with good spatial locality on the send buffer matrix). However, P1 has non-linear memory load patterns on the send buffers, exhibiting poor spatial locality. This shows that the memory copy ordering on the receive buffer matrix (and hence, in the send buffer matrix) is important to achieve good spatial locality during the Alltoall operation.

Cache-oblivious algorithms [4] have been proposed for transposing matrices in prior work. They recursively subdivide the matrix into smaller blocks, perform local operations in a cache-efficient way, and merge the results to get the final transpose. Examples of cache-oblivious algorithms include space-filling curves such as Hilbert [5] curves, which have been known to preserve locality well for a one-dimensional traversal of a matrix and have been used to optimize dense matrix operations [6]. This makes these curves an excellent choice for achieving high spatial locality. However, in a distributed setting such as MPI_Alltoall, a different memory copy ordering could perform better for certain message sizes/process counts due to the effect of cache coherence protocols, available bus bandwidth, how buffer addresses are laid out in memory, and the mechanism of sharing data between processes. *This motivates the need for designs that can adapt memory copy ordering to different architectures and memory layouts while achieving good cache performance.*

Memory bandwidth plays an important role in the performance of an MPI_Alltoall as the number of memory transactions of an MPI_Alltoall increases by a factor of N^2, where N is the number of processes. Improving cache bandwidth for MPI_Alltoall on multiple processes per node is challenging as the number of memory transactions significantly increases, and not all data can fit into the cache. Many modern CPUs provide non-temporal instructions, which give a hint to the processor that access to the memory can bypass the cache, thereby leading to fewer memory transactions on the bus. This can significantly improve bandwidth for MPI_Alltoall-like operations due to a reduction in bus transactions. However, using these instructions comes with a latency hit, potentially evicts data blocks in the cache (depending on the processor implementation), or leads to sub-par temporal locality due to important data not being written to the cache. Prior work [7] has explored the use of non-temporal instructions for a two-copy (shared-memory) paradigm in MPI. However, their approach has different access patterns due to the two-copy paradigm (Section V-E) and does not target Alltoall operations. *This motivates the need for Alltoall designs that can retain some blocks of data in the cache while also using non-temporal instructions to reduce bus transactions in bandwidth-constrained scenarios.*

In this paper, we design an Intra-node Alltoall algorithm that dynamically switches between memory copy orderings and uses non-temporal instructions to reduce transactions on the bus. Our approach significantly reduces latency and provides higher bandwidth when compared to state-of-the-art solutions. To this end, we propose HINT: An algorithm for Cache-Efficient MPI_Alltoall using Hybrid Memory Copy Ordering and Non-Temporal Instructions.

II. Motivation

A. Increased cache usage of dense collectives

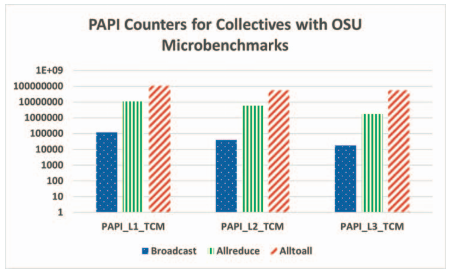


Figure 2: L1, L2, and L3 total cache misses with PAPI for different collectives on an Intel Xeon Gold 6138 CPU.

We motivate the need for cache-friendly designs for alltoall by profiling cache misses using the Performance Application Programming Interface (PAPI) [8] for three collective communication patterns in OSU micro-benchmarks [9]. We chose broadcast, allreduce, and alltoall as candidate patterns as they are in increasing order of the number of overall memory transactions per process. Figure 2 shows L1, L2, and L3 misses on rank 0, running our candidate patterns for 32 processes on an Intel Xeon Gold 6128 CPU. The processes are spread across both sockets, with the first 16 ranks on socket one and the rest on socket two. As shown in the figure, the allreduce pattern has around 100× the number of cache misses over broadcast at all levels, and the alltoall pattern has up to 10× the cache misses over allreduce. While the number of cache misses depends on the underlying algorithm involved, the experiment above shows a general trend – denser communication patterns tend to have a significantly higher number of cache misses and, hence, are more likely to cause cache thrashing. Since keeping as much data at a time in caches is known to have significant performance advantages, having better cache locality for dense collectives is a critical problem to solve.

B. Bandwidth advantages of non-temporal instructions

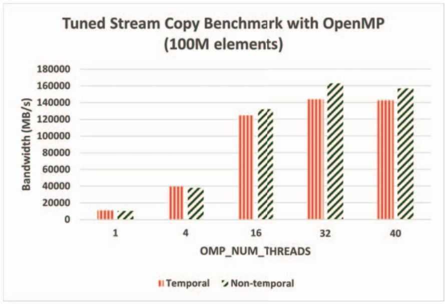


Figure 3: Tuned Stream benchmark copy bandwidth results for 100M doubles comparing non-temporal stores and regular stores for different thread counts on an Intel Xeon Gold 6138 CPU.

We use a modified/tuned version of the STREAM copy benchmark [10] to show the bandwidth advantages of non-temporal instructions. We modified the *tuned_STREAM_Copy* function to use *_mm512_stream_pd* non-temporal store intrinsic for writing into the result buffer, with each iteration of the loop copying 64 bytes of data. Figure 3 shows the bandwidth in MB/s for a memory copy of 100M elements by varying the number of OpenMP threads on an Intel Xeon Gold 6138 CPU. While non-temporal stores show a drop in bandwidth for fewer threads (1-4), we observe an increase in memory bandwidth when increasing the thread count. This is primarily due to the increase in memory traffic when the number of threads increases. Non-temporal stores avoid read-for-ownership (RFO) bus requests and do not disturb the cache hierarchy, which results in improved bandwidth when the memory copies are bandwidth-limited. This shows that non-temporal stores can improve large message copy performance when the number of cores participating in memory copies is large. Incorporating non-temporal instructions in MPI_Alltoall requires a fine-grained approach for a given cache hierarchy and buffer access pattern.

III. Contributions

The overall idea of this paper is to dynamically switch between different memory copy orders (or traversals of the matrix of receive buffers) for more efficient cache behavior when performing Alltoall. For large messages, we use non-temporal instructions to accelerate copies in bandwidth-constrained scenarios (for instance, at higher core counts). We use x86 architectures and the AVX instruction set extensions as a case study for non-temporal instructions, but our designs are extendable and can be tuned for future architectures. In summary, our paper makes the following key contributions:

1) Identify and highlight trends in cache misses for dense collective patterns such as MPI_Alltoall.
2) Evaluate the impact and potential bandwidth benefits of non-temporal instructions in multi-core scenarios and design a hybrid memory copy scheme that combines the benefits of non-temporal and regular temporal instructions. Our memory copy scheme shows up to a **50%** reduction in L1/L2/L3 cache misses and a **10x** reduction in processor stalls waiting for memory writes for a 1MB MPI_Alltoall on 32 processes.
3) Analyze the effect of ordering memory copies for Alltoall and propose a hybrid design that dynamically switches between different memory copy orders. We observe up to a **20%** improvement by switching between memory copy orderings versus fixing one memory copy ordering for all message sizes.
4) Discuss extensions to the proposed Alltoall algorithm for non-powers-of-two process counts/multi-node systems.
5) Design a hybrid scheme that features non-temporal instructions in MPI_Alltoall and combines them with the proposed memory copy orders to further accelerate large message ($\geq$64K) scale-up performance. The hybrid scheme outperforms pure non-temporal/temporal memory copies by up to 20% in MPI_Alltoall micro-benchmarks.
6) Perform a thorough evaluation of the performance of the proposed designs against state-of-the-art MPI libraries on different architectures. Our designs outperform state-of-the-art by up to **10x** for MPI_Alltoall evaluated using OSU micro-benchmarks and up to **22.2**% for the P3DFFT application.

IV. Background

A. XPMEM

Linux Cross-Memory Attach (XPMEM) [11] is a Linux kernel module that enables a process to access the memory regions of other processes. This is done by exposing a part of the virtual memory address space of the owner process and attaching that memory region to the process that is accessing it. The owner process can expose its region by calling *xpmem_make()* and other processes can attach this region by calling *xpmem_get()* and *xpmem_attach()*. The attached memory can simply be accessed using direct loads and stores. XPMEM is particularly beneficial for applications that require low-latency access to remote memory. Many MPI implementations leverage XPMEM to optimize intra-node communication by performing single-copy transfers.

B. Cache-oblivious Algorithms

Cache-oblivious algorithms are designed to efficiently utilize a processor's cache without requiring knowledge of the cache size or organization. An algorithm is cache-oblivious if the program variables are independent of the hardware configuration. The aim is to minimize cache misses and exploit cache performance, making these algorithms optimal in an asymptotic sense and suitable for a wide range of memory hierarchies without specific tuning for any particular cache size or machine. Examples of cache-oblivious algorithms can be found in matrix multiplication, matrix transposition, sorting, and FFT applications [4].

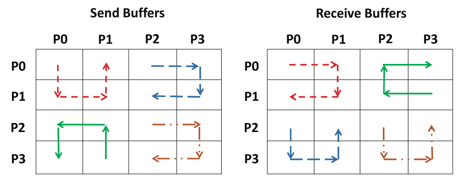
V. Design and Implementation

In this section, we first show the implementation of different one-dimensional traversals of a matrix. Then, we extend these traversals to a distributed setting for use in MPI_Alltoall and identify bottlenecks for large message sizes. Finally, we propose a solution to overcome cache-related bandwidth bottlenecks.

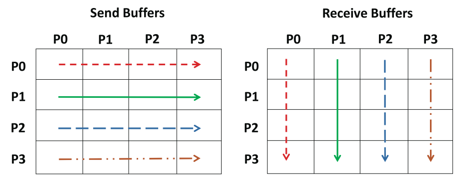
A. *Implementing different matrix traversals for use in Alltoall*

An MPI_Alltoall can be viewed as a global matrix transpose. The send/receive buffer on each process is contiguous and contains data to be sent/received to/from every other process. The set of send buffers and the set of receive buffers can be visualized as two matrices. The goal of the Alltoall is to transpose the send buffers into the receive buffers. As a first step, we implement three different one-dimensional traversals of a matrix - Hilbert curves, row-wise and column-wise. Each one is labeled based on the order of traversal of the receive buffer matrix. We chose Hilbert curves as they are known to be cache-oblivious and slightly better performing than other space-filling curves for many HPC patterns [12]. The motivation for selecting row-wise and column-wise traversals is simple – the row-wise order accesses memory addresses in the receive buffer matrix linearly, whereas the column-wise order performs linear access of memory addresses in the send buffer matrix. We only designed for process grids whose size is a power of two since Hilbert curves are more natural to implement on power of two grids.

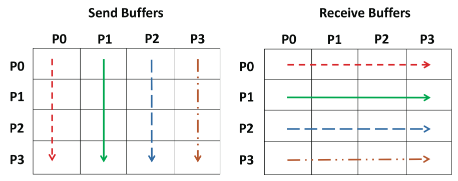
Figure 4 shows a visualization of the three different orders. The Hilbert curve recursively divides the matrix into sub-matrices of equal size. Each 'base' case sub-matrix contains a 'U' shape. These are rotated based on different orientations to form the final Hilbert curve. The row-wise order scans the receive buffers row by row, whereas the column-wise order scans in column order. Once a one-dimensional order is generated (using numbers starting from 1 up to N^2 where N is the number of processes), we subdivide the matrix to distribute tasks to different processes. This varies based on the traversal order. Algorithm 1 shows the methodology behind distributing memory copy operations between processes. The functions return the bounds of a sub-matrix, which specifies the part of the receive buffer matrix a process will work on. In the row-wise case (get_bounds_row), the row corresponding to the receive buffer for a rank (a numeric identifier for each MPI process) is returned. In the column-wise case (get_bounds_col), the algorithm returns a column in the receive buffer matrix corresponding to the rank. For Hilbert curves, the algorithm divides the matrix into equal-sized sub-matrices. This is shown in the get_bounds_hilbert function. We fix the number of rows for each sub-grid by taking the largest power of two greater than or equal to the square root of N, where N is the number of processes. Once the number of rows is fixed, we divide N by that number to get the number of columns in each sub-grid. Each sub-grid contains the work to be done by a process during the Alltoall. As the processes are equally divided across processor sockets (also known as a spread-based mapping strategy), we first chose to fix the rows in the sub-matrix to preserve NUMA locality.



(a) Hilbert curve based ordering



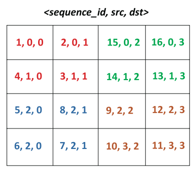
(b) Row-wise ordering



(c) Column-wise ordering

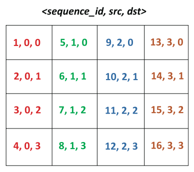
Figure 4: Ordering of memory copy operations on every process for Hilbert curves, row-wise and column-wise traversals of the receive buffer matrix.

B. *Generating metadata for the Alltoall operation*


<sequence_id, src, dst>

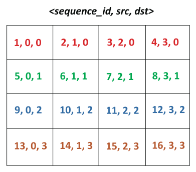
1, 0, 0	2, 0, 1	15, 0, 2	16, 0, 3
4, 1, 0	3, 1, 1	14, 1, 2	13, 1, 3
5, 2, 0	8, 2, 1	9, 2, 2	12, 2, 3
6, 2, 0	7, 2, 1	10, 3, 2	11, 3, 3

(a) Hilbert ordering


<sequence_id, src, dst>

1, 0, 0	5, 1, 0	9, 2, 0	13, 3, 0
2, 0, 1	6, 1, 1	10, 2, 1	14, 3, 1
3, 0, 2	7, 1, 2	11, 2, 2	15, 3, 2
4, 0, 3	8, 1, 3	12, 2, 3	16, 3, 3

(b) Column ordering


<sequence_id, src, dst>

1, 0, 0	2, 1, 0	3, 2, 0	4, 3, 0
5, 0, 1	6, 1, 1	7, 2, 1	8, 3, 1
9, 0, 2	10, 1, 2	11, 2, 2	12, 3, 2
13, 0, 3	14, 1, 3	15, 2, 3	16, 3, 3

(c) Row-wise ordering

Figure 5: Metadata tuple grid generated in Algorithm 2 for three different memory copy orderings.

Once the receive buffer matrix is divided based on Algorithm 1, we must process the grid to add metadata for the Alltoall operation. The idea is to have metadata that provides each process information on the data transfers that need to happen.

Algorithm 2 describes this in detail. In the first step, we take the matrix from generating different orders and generate a three-element tuple for each entry. This is shown in the generate_tuple_grid function. For each element in the matrix,

Algorithm 1: Algorithm to distribute work between processes for different memory copy orders for the Alltoall operation

```
Input : N where N is a power of two
Output: Grid with numbers in a specific matrix traversal order
Function: get_bounds_hilbert (grid, rank, N)
begin
    $num\_subgrid\_rows \leftarrow next\_pof2(sqrt(N))$
    $num\_subgrid\_cols \leftarrow N/num\_subgrid\_rows$
    $X\_start \leftarrow 0, X\_end \leftarrow num\_subgrid\_rows$
    $Y\_start \leftarrow 0, Y\_end \leftarrow num\_subgrid\_cols$
    $ctr \leftarrow 0$
    while $ctr$ < rank do
        if Y_end != N then
            $Y\_start \leftarrow Y\_start + num\_subgrid\_cols$
            $Y\_end \leftarrow Y\_start + num\_subgrid\_cols$
        end
        else
            $Y\_start \leftarrow 0, Y\_end \leftarrow num\_subgrid\_cols$
            $X\_start \leftarrow X\_start + num\_subgrid\_rows$
            $X\_end \leftarrow X\_start + num\_subgrid\_rows$
        end
        $ctr \leftarrow ctr + 1$
    end
    return X_start, X_end, Y_start, Y_end
end
Function: get_bounds_row (grid, rank, N)
begin
    X_start = rank, X_end = rank Y_start = 0, Y_end = N - 1 return
    X_start, X_end, Y_start, Y_end
end
Function: get_bounds_col (grid, rank, N)
begin
    X_start = 0, X_end = N - 1 Y_start = rank, Y_end = rank return
    X_start, X_end, Y_start, Y_end
end
```

Algorithm 2: Algorithm to generate metadata for the Alltoall operation

```
Input : N where N is a power of two
Input : rank- An identifier for each process in the MPI context
Output: Grid with numbers in a specific matrix traversal order
Function: generate_tuple_grid (grid, N)
begin
    $tuple\_grid \leftarrow malloc(N * N * 3 * sizeof(int))$
    for $r \leftarrow 0$ to $N$ do
        for $c \leftarrow 0$ to $N$ do
            $tuple\_grid[r][c] \leftarrow (grid[r][c], r, c)$
        end
    end
end
Function: gen_metadata_encoded (grid, rank, N)
begin
    $tuple\_grid \leftarrow generate\_tuple\_grid(grid, N)$
    $bounds \leftarrow get\_bounds(tuple\_grid, rank, N)$;
    $order \leftarrow get\_flattened\_submatrix(bounds)$
    $sort(order, key\_index \leftarrow 0)$
    return order
end
```

the tuple encodes three different values — 1) a number that represents the order in which data must be accessed (larger numbers are accessed later in the traversal), 2) the row, which represents the source rank, and 3) the column, which represents the destination rank. The goal of having a tuple is for each process to know the source and destination buffers it must do memory loads/stores on.

Figure 5(a) shows a 4X4 grid of tuples subdivided into four equal-sized grids (of size 2X2) for the Hilbert curve-based order. Each color represents work to be done for a certain rank in the Alltoall – red for rank 0, green for rank 1, blue for rank 2, and orange for rank 3. For example, consider the sub-grid in red, which corresponds to the work for rank 0. Flattening this sub-grid yields four entries – (1,0,0), (2,0,1), (4,1,0), (3,1,1). Once the sub-grid is flattened, we sort the tuples using the first tuple element as the key. This gives the list – (1,0,0), (2,0,1), (3,1,1), (4,1,0), which is in Hilbert order for the sub-grid. Rank 0 would then use the list to determine the exact order of operations it has to execute for the Alltoall. This is described in V-C. Similarly, a metadata grid is generated for the other two traversals, as shown in Figures 5(c) and 5(b). This metadata is saved inside the communicator, so order/metadata generation only happens once per communicator.

C. Algorithm for XPMEM-based Alltoall using metadata

The tuple_grid obtained from the previous step is used to implement the Alltoall in MPI. We use XPMEM to map remote process address spaces so that any arbitrary process can perform copies on behalf of another process.

Algorithm 3 shows our algorithm implementation. The Algorithm takes the ordering of tuples described in the previous section. We cache this inside a communicator to avoid the repeated generation of the Hilbert curve and the tuple grid. In the first step of the algorithm, all processes write metadata about the send/receive buffer address so that they can be used by other processes for address space mapping using XPMEM. Then, every process iterates over the order array (whose size equals the number of local processes in the communicator) to perform memory copies. The order tuple contains the intended source and destination ranks for each loop iteration. This is then translated to a global rank for use in the case of multi-node alltoalls for correct indexing of the buffers. If the source/destination are remote processes, the process tries to attach to the remote buffer using XPMEM. This involves a kernel call to map the remote process's address space to the local process but is amortized using a cache that avoids repeated calls for the same buffer. Once the correct pointers are obtained and the offsets are calculated, the process issues a memory copy from source to destination. At the end of the algorithm, all processes execute a barrier. This ensures correctness as it mandates every process to notify completing work on behalf of other processes.

We implemented both an offline and online approach for letting the algorithm pick between different memory orderings. In the offline approach, we run the MPI library with an Alltoall benchmark with warmup iterations (so that setup costs are ignored) for various process counts and message sizes. We then generate a JSON file, which the MPI library can read at runtime. This information is then used to select the ideal memory copy ordering for a given message size/process count. The advantage of this is that the code does not need iterations during calls to MPI_Alltoall to try different combinations before finalizing the best memory ordering for a given message size. In the online approach, we let each memory copy ordering run three times back to back for each message size, process count and send/receive buffer address combination (ignoring the first iteration in the Alltoall, which involves setup costs). Once one memory copy ordering is run, we switch to the next one. At the end of nine iterations (since we have three different memory orderings in our current implementation), the code picks the memory copy ordering with the lowest time. The number of times each memory ordering is run is a parameter that can be changed if required. All evaluations in this paper are done with the offline approach since this only needs to happen once for a given architecture.

Algorithm 3: XPMEM-based algorithm using any generic ordering for MPI_Alltoall

```
Input  : sendbuf: Send buffer used in the alltoall
Input  : recvbuf: Receive buffer used in the alltoall
Input  : nbytes: Size of buffer to be communicated
Input  : order: List of tuples containing orders from the metadata step
Input  : comm_ptr: Communicator used by MPI for the alltoall
Input  : shmem_comm_ptr: Communicator for intra-node operations
Output: Performs a distributed matrix transpose
Function: memory_copy_order_alltoall (sendbuf, recvbuf, nbytes,
order, shmem_comm_ptr, comm_ptr)
begin
    $share\_xpmem\_metadata(sendbuf)$
    $share\_xpmem\_metadata(recvbuf)$
    $local\_size = get\_num\_procs\ (shmem\_comm\_ptr)$
    $rank = get\_rank(comm\_ptr)$
    for $i \leftarrow 0$ to $local\_size$ do
        $src \leftarrow get\_global\_rank(comm\_ptr, order[i].src)$
        $dst \leftarrow get\_global\_rank(comm\_ptr, order[i].dst)$
        $src\_buf \leftarrow sendbuf$
        $dst\_buf \leftarrow recvbuf$
        if $src$ != $rank$ then
            $src\_buf \leftarrow get\_remote\_sendbuf(src)$
        end
        if $dst$ != $rank$ then
            $dst\_buf \leftarrow get\_remote\_recvbuf(dst)$
        end
        $send\_offset \leftarrow nbytes * src$
        $recv\_offset \leftarrow nbytes * dst$
        $src\_buf \leftarrow src\_buf + send\_offset$
        $dst\_buf \leftarrow dst\_buf + recv\_offset$
        $memcpy(dst\_buf, src\_buf, nbytes)$
    end
    $intra\_node\_barrier(shmem\_comm\_ptr)$
end
```

D. Empirically analyzing trends with different orderings for MPI_Alltoall

In this section, we analyze trends using the proposed orders on a range of message sizes on an Intel Cascade Lake processor. Figure 6 shows latency numbers for MPI_Alltoall with the OSU micro-benchmark suite on 8, 16, and 32 processes. We split the analysis into two message ranges – the first covering messages up to 32K bytes (around the L1 cache size) and the second from 32K bytes to 1M (the L2 cache size).

Figures 6(a), 6(b) and 6(c) show latency numbers up to 32K bytes. Our Hilbert curve-based implementation performs up to 20% worse than row-wise or column-wise traversals up to a message size of 16K for 8/16PPN and up to 8K for 32PPN. For 8PPN, the column-wise traversal outperforms all orderings

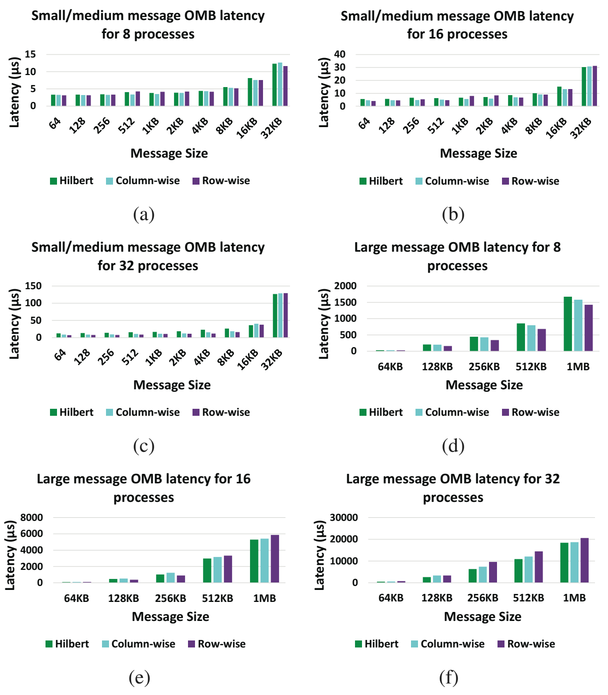


Figure 6: MPI_Alltoall performance comparisons between the three proposed memory copy ordering schemes using OSU micro-benchmarks for different processes per node (PPN) on an Intel Xeon Platinum processor.

up to 2K bytes, after which the row-wise ordering performs better. However, at 32PPN, the row-wise ordering outperforms the column-wise ordering up to 8K bytes. This experiment shows that one order for accessing the receive buffer does not win in all cases, which demonstrates the need for a hybrid design that switches between different schemes based on the message size and PPN.

Figures 6(d), 6(e) and 6(f) show latency numbers from 64K bytes to 1M bytes. The row-wise ordering outperforms column-wise and Hilbert schemes by up to 10% for 8PPN. For 16PPN and 32PPN, the trend seen in the small message spectrum is reversed. The Hilbert curve outperforms the column-wise and row-wise ordering except for 128K bytes and 256K bytes at 16PPN. On running profiling with PAPI for 128K bytes and 256K bytes, we found that the row-wise scheme had a significantly lower value for PAPI_STL_ICY, which shows the number of cycles with no instruction issue, indicating that the row-wise scheme more efficiently sent in micro-ops to the processor front-end. We are still investigating further to see why such trends occur.

Another interesting trend we observed when the message size crosses the L1 cache size is shown in Figure 7. As shown in the figure, the value of PAPI_MEM_WCY jumps four-fold from 16K to 32K and sharply increases as the message size increases. The PAPI_MEM_WCY represents the number of cycles stalled waiting for memory writes. On the Intel Cascade Lake processor, this counter maps to the

RESOURCE_STALLS:SB. This indicates that the processor core was stalled for many cycles, waiting for the store buffer to be free. A store buffer exists between the CPU and core local caches to speculate on stores on an invalidated cache line so that the core does not have to stall until an acknowledgment is received [13]. Since the number of store buffers is limited, this motivates the need for designs that can reduce the number of cache transactions on the bus, thereby improving bandwidth and reducing stalls.

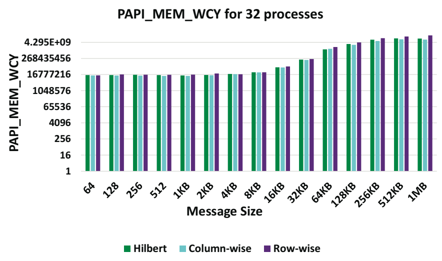


Figure 7: Profiling store buffer-related memory resource stalls using the PAPI_MEM_WCY counter from PAPI for a 32 process MPI_Alltoall on an Intel Xeon Platinum system. We observe a steep increase in memory stalls starting from 32K bytes.

E. Obtaining better bandwidth with non-temporal stores

The number of memory transactions for Alltoall increases significantly with the increase in message size and the number of processes. Since the operation involves a matrix transpose, memory transactions might require writes to caches that must invalidate other caches before proceeding further. This places a read-for-ownership (RFO) request on the bus. In large message Alltoalls, most of the data communicated does not fit in the cache, and hence RFO requests could potentially waste bandwidth and cause stalls in the memory pipeline. This is observed in the previous section where PAPI_MEM_WCY sharply increases beyond 16K bytes. To alleviate this bottleneck, non-temporal instructions are provided in many architectures to give hints to a processor that a memory region does not have to be cached. This significantly increases bandwidth but has a potential hit in latency for smaller message sizes.

To overcome these bottlenecks, we propose a hybrid memory copy for our Alltoall algorithm. Algorithm 4 shows our proposed design. When the size of the message is less than a given threshold, we fall back to a regular memcpy. If the message size to be copied is greater than or equal to the threshold, a regular memcpy is called up to the threshold, and the algorithm uses a non-temporal memory copy with prefetched loads for the rest of the message. The threshold is chosen on a per-communicator basis (since it changes based on the process count/architecture). Figure 8 shows a comparison between temporal and non-temporal instructions on 32 processes on both an AMD and an Intel Cascade Lake system. Temporal memory copies significantly outperform non-temporal instructions up to 128KB on the AMD system and 64KB on the Intel system. However, beyond these thresholds, the non-temporal copy is much faster than the temporal copy. Our scheme uses the best of both temporal and non-temporal instructions by falling back to temporal copies for messages less than the threshold and then dynamically adjusting the ratio of temporal to non-temporal instructions for messages larger than the threshold. Figure 9 shows how we use the memory copy with the Hilbert ordering. Each 'block' corresponds to one load/store operation in the receive buffer matrix. The processes follow the order defined in the Alltoall and use a combination of temporal and non-temporal stores.

We empirically found that the ideal temporal copy size depends on the number of processes and the size of the L2 cache. The initial value of threshold T is determined by dividing the L2 cache size and the number of processes. Intuitively, this allows the memory copy to use the L2 cache effectively and reduces excessive bus transactions using non-temporal instructions. This leads to significant gains in performance. However, for very large messages (>= L2 cache) on some scales, we observed that only using non-temporal instructions yielded slightly better performance, likely due to the bus bandwidth being the dominating factor for latency. Hence, our design tries to scale down the threshold by a factor of two in every iteration to see if performance improves before deciding on a final value for an Alltoall. This is achieved by the get_runtime_threshold function in Algorithm 4.

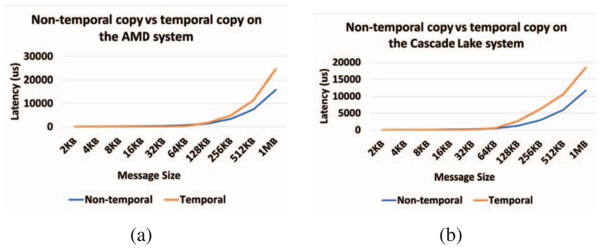


Figure 8: Comparing non-temporal and temporal memory copies for a 32PPN MPI_Alltoall using OSU micro-benchmarks.

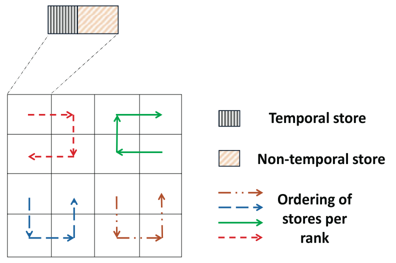
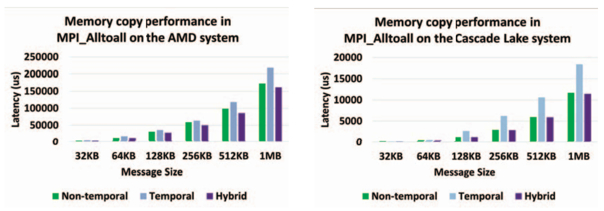


Figure 9: Proposed hybrid non-temporal copy design using a Hilbert-curve based ordering.

Figure 11 shows the efficacy of our non-temporal designs for Alltoall on an Intel Cascade Lake system for 16 cores and 32 cores. The figure shows numbers with Hilbert curves but should apply to other orders as well. As shown in the figure, our temporal and non-temporal hybrid store design significantly reduces stalls waiting for the store buffer and cache misses across the cache hierarchy.

Authors in [7] propose adaptive memory copy schemes for optimizing MPI collectives on shared memory multi-cores.

While their work also uses non-temporal stores, it follows the two-copy (shared-memory) paradigm and does not target alltoall. In a two-copy paradigm, every process copies a chunk of data into shared memory, and remote processes copy data from shared memory into their receive buffers (also known as copy-in-copy-out). This allows for temporal locality (since buffers copied to shared memory will be accessed again for collectives like Allgather) but incurs an additional copy. Their adaptive memory copy scheme (Algorithm 1 in their paper) uses a flag to determine if data is likely to be accessed again or not and either uses a temporal copy or a non-temporal copy. However, this does not apply to our use case since we use XPMEM (a single-copy scheme), due to which data is never accessed again within the Alltoall. Moreover, in contrast to their solution, our scheme for memory copies combines both temporal instructions and non-temporal instructions. Figure 10 shows up to 20% improvements when combining both instructions (using the hybrid copy scheme) for message sizes near the cache boundary when compared against pure temporal/non-temporal instructions in an intra-node MPI_Alltoall. For other message sizes, we observed improvements between 5-10% in a few cases and equal performance to the best of temporal/non-temporal instructions for other sizes.

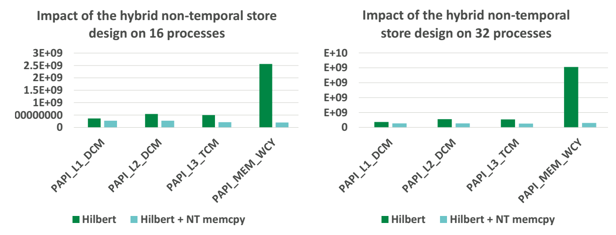


(a) 128PPN - AMD (b) 32PPN - Intel

Figure 10: Hybrid copy compared against using just non-temporal/temporal memory copies in the Alltoall OSU micro-benchmark.

Algorithm 4: Customized memory copy using non-temporal instructions

```
Function: hybrid_nt_memcpy (recvbuf, sendbuf, nbytes, T)
begin
    if nbytes < T then
        | memcpy(recvbuf, sendbuf, nbytes)
    end
    else
        | R = get_runtime_threshold(nbytes, T)
        | memcpy(recvbuf, sendbuf, R)
        | nt_memcpy(recvbuf, sendbuf, nbytes− R)
    end
end
```



(a) 16 processes (b) 32 processes

Figure 11: Impact of hybrid non-temporal copy design for a 1M byte MPI_Alltoall on an Intel Cascade Lake processor. We measure cache misses for all three cache levels and memory stalls using PAPI counters inside OSU-microbenchmarks.

Algorithm 5: Algorithm extending memory copy aware alltoall for use in networked systems

```
Input  : sendbuf: Send buffer used in the alltoall
Input  : recvbuf: Receive buffer used in the alltoall
Input  : nbytes: Size of buffer to be communicated
Input  : order: List of tuples containing orders from the metadata step
Input  : comm_ptr: Communicator used by MPI for the alltoall
Output: Performs a distributed matrix transpose
Function: mpi_alltoall (sendbuf, recvbuf, nbytes, order, comm_ptr)
begin
    /* —— Intra-node step —— */
    shmem_comm_ptr ← get_intra_node_comm(comm_ptr)
    memory_copy_order_alltoall(sendbuf, recvbuf, nbytes,
    order, shmem_comm_ptr, comm_ptr)
    /* —— Inter-node step —— */
    global_size = get_num_procs(comm_ptr)
    rank = get_rank(comm_ptr)
    for i ← 0 to global_size do
        src ← (rank − i + global_size)%global_size
        dst ← (rank + i)%global_size
        src_buf ← sendbuf + dst ∗ nbytes
        dst_buf ← recvbuf + src ∗ nbytes
        requests[2] ← init_requests()
        if !(is_in_same_node(src, rank) then
            | isend(src_buf, nbytes, dst, comm_ptr, &requests[0])
        end
        if !(is_in_same_node(dst, rank) then
            | irecv(dst_buf, nbytes, src, comm_ptr, &requests[1])
        end
        waitall(&requests)
    end
end
```

F. Extensions to the algorithm for use in networked systems and non-powers of two process counts

Extentions for networked systems- Algorithm 5 shows how the memory copy order alltoall (Algorithm 3) can be used in a generic alltoall that works for networked cases as well. The first phase involves an intra-node step, which calls the proposed algorithm on a communicator representing each node's set of processes. Once the intra-node step is done, a pairwise alltoall exchange (or any other inter-node algorithm) can be implemented by skipping sends/receives from processes within the same node. Note that further optimizations are possible here, such as pipelining intra-node operations with inter-node operations, but that remains out of the scope of this paper. Our solution can also be used by applications that create multiple communicators. One example is FFT-based applications (like P3DFFT), which create row and column communicators. These communicators can be created such that each column contains all processes within the node, and each row contains processes across different nodes. This way, the "column-wise" alltoall would use our algorithm, and the "row-wise" alltoall potentially uses something different.

Extensions for non-powers of two: The algorithms assume process counts that are a power of two only due to using a standard geometric representation of the hilbert curve ($2^k \cdot 2^k$ lattice for a k-order curve) and because most applications tend to use powers of two process counts. Our approach is generic and can accept extensions to the hilbert curve for non-powers of two square edges ([14]), or other space-filling curves that do not have this limitation, and can hence be extended to include non-powers of two. Only Algorithm 1 needs to be changed to incorporate the new work distribution between processes. Another potential solution to supporting non-powers of two process counts is using a Hilbert curve for the square region with edge size a power-of-two in the alltoall matrix and filling the rest of the square with a different traversal (either a space-filling curve or any other ordering).

VI. Experimental Evaluation

We evaluate our designs on three different x86 architectures that support the AVX instruction set. The first system contains two Intel Xeon Platinum 8280 Cascade Lake CPUs (one per socket), the second system has two sockets with Intel Xeon Gold 6138 Skylake CPUs, and the third system has AMD EPYC 7713 CPUs with two sockets and 128 cores. The detailed configuration of these systems is shown in Table I.

On each system, we compare the performance of our design for MPI_Alltoall against state-of-the-art libraries on all systems – MVAPICH2-X v2.3/Intel-MPI 19.0.9 for the first system with Cascade Lake CPUs and MVAPICH2-X v2.3/HPC-X v2.16 for the second and third systems. For Intel-MPI and HPC-X, we use the default modules provided on the system for comparisons. We use OSU micro-benchmarks for demonstrating benchmark-level results. Each benchmark run is an average of five trials, with each one running for the default number of iterations set by the benchmark. On the AMD system, we do not present results for 128ppn due to anomalies we experienced when testing all MPI implementations; we will investigate these outliers. For application-level results, we demonstrate improvements with P3DFFT [15] by taking an average of five runs for the 'CPU time per loop' metric.

Table I: Hardware specifications of clusters

Specification	**AMD EPYC**	**Intel Platinum**	**Intel Gold**
Processor Family	AMD EPYC	Intel Cascade Lake	Intel Skylake
Processor Model	EPYC 7742	Xeon Platinum 8280	Xeon Gold 6138
Max Clock Speed	3.72 GHz	4 GHz	3.7 GHz
Sockets	2	2	2
Cores Per socket	64	28	20
NUMA nodes	2	2	6
RAM (DDR4)	256 GB	192 GB	192 GB
AVX capability	AVX2	AVX-512	AVX-512

A. Micro-Benchmark Evaluation - Small/Medium messages

We first evaluate our designs against state-of-the-art libraries for small/medium message sizes. We have only included numbers where the number of processes is a power of two since our Hilbert curve implementation assumes that process grid dimensions are powers of two. Figures 12, 13 and 14 show results for our proposed design starting from 256 bytes to 32K.

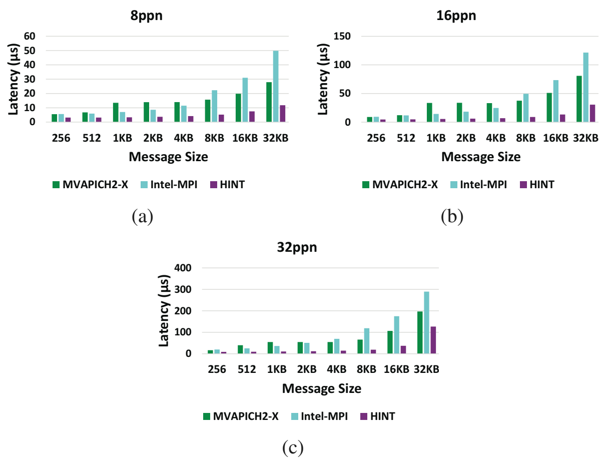


Figure 12: Performance comparison of HINT against Intel-MPI and MVAPICH2-X on the Intel Xeon Platinum system (Cascade Lake) for message sizes from 256 bytes to 32K bytes using OSU micro-benchmarks.

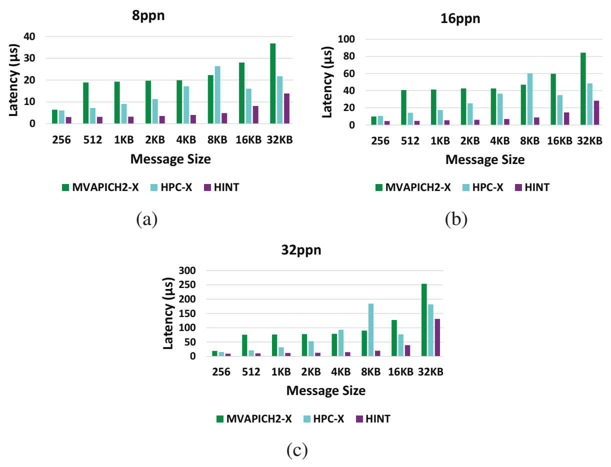
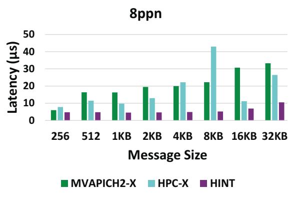
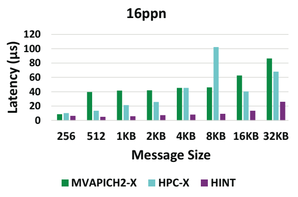


Figure 13: Performance comparison of HINT against HPC-X and MVAPICH2-X on the Intel Xeon Gold system (Skylake) for message sizes from 256 bytes to 32K bytes using OSU micro-benchmarks.

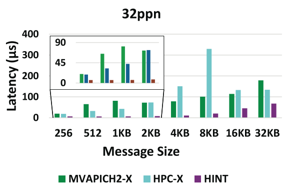
On the Cascade Lake system (Figure 12), our proposed designs outperform both Intel-MPI and MVAPICH2-X up to at least **67**% for 8PPN, **75**% for 16PPN, and **76**% for 32PPN. We attribute this to better cache locality and reduced cache coherence transactions using different memory copy orderings. On the Skylake system (Figure 13), we observe similar trends with our proposed design outperforming HPC-X and MVAPICH2-X up to at least **78**% for 8PPN, **81**% for 16PPN, and **81**% for 32PPN. On the AMD system, which has AVX2 instructions and a different cache coherence protocol, we still see benefits across all PPNs. As shown in Figure 14, we see up to at least **90**% improvement compared to HPC-X and MVAPICH2-X for 64PPN with similar results for



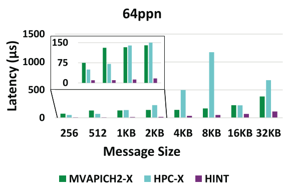
(a)



(b)



(c)



(d)

Figure 14: Performance comparison of HINT against HPC-X and MVAPICH2-X on the AMD system for message sizes from 256 bytes to 32K bytes using OSU micro-benchmarks.

other PPNs. This shows that our design can adapt to different architectures and applies to various message sizes and process counts.

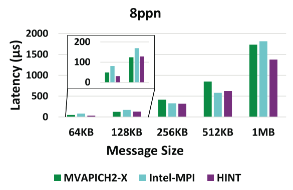
B. Micro-Benchmark Evaluation - Large messages

Our designs use a combination of memory copy ordering and non-temporal instructions for large messages to get better bandwidth and latency. Similar to our small message comparisons, we evaluate against state-of-the-art libraries on all three platforms. Figures 15, 16 and 17 show results for our proposed design starting from 64K to 1M bytes.

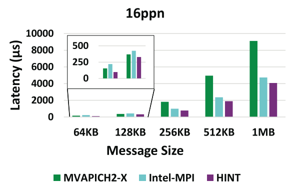
On the Cascade Lake system (Figure 15), we see up to **37**% for 8PPN, **37**% for 16PPN and **30**% for 32PPN against Intel-MPI. As discussed in the analysis section, using non-temporal instructions eliminates RFO transactions and significantly reduces memory stalls/store misses, which is why we see improved performance. Similarly, on the Skylake system (Figure 16) we outperform HPC-X and MVAPICH2-X by up to **22**% for 8PPN, **36**% for 16PPN, and **23**% for 32PPN. The Intel architectures are AVX-512 compatible and have 512-bit wide SIMD units for memory/compute operations. The AMD system, on the other hand, is AVX2 capable, so it only has a 256-bit wide SIMD unit. Even in the AMD case, we see improvements of up to **73**% starting from 8PPN to 64PPN. This shows that our non-temporal copy designs apply to architectures other than the one we analyzed. While we observed a couple of data points with anomalies that have to be investigated (for instance, 1MB messages on the AMD system for 32PPN and the Skylake system for 16PPN), the trends in latency for different message sizes are largely the same across all architectures.

C. Application-level evaluation with P3DFFT

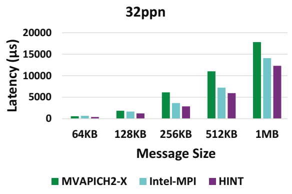
Fourier transforms are widely used across various scientific domains and are often bottlenecked by communications when performing distributed matrix transposes (MPI_Alltoall).



(a)

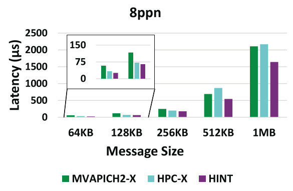


(b)

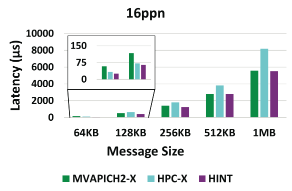


(c)

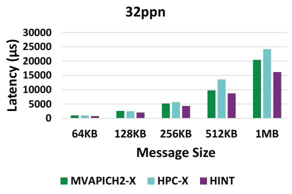
Figure 15: Performance comparison of HINT against Intel-MPI and MVAPICH2-X on the Intel Xeon Platinum system (Cascade Lake) for message sizes from 64K bytes to 1M bytes using OSU micro-benchmarks.



(a)

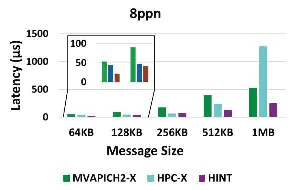


(b)

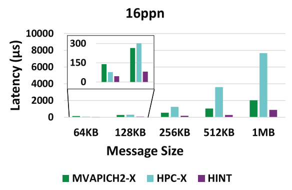


(c)

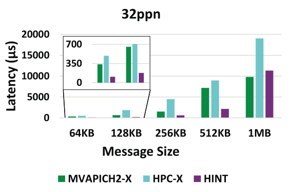
Figure 16: Performance comparison of HINT against HPC-X and MVAPICH2-X on the Intel Xeon Gold system (Skylake) for message sizes from 64K bytes to 1M bytes using OSU micro-benchmarks.



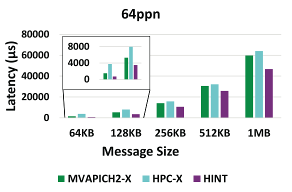
(a)



(b)

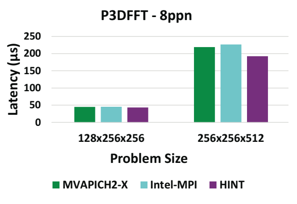


(c)

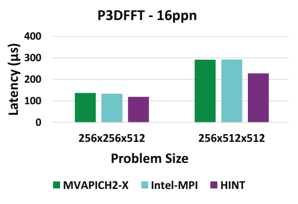


(d)

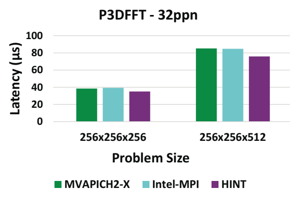
Figure 17: Performance comparison of HINT against HPC-X and MVAPICH2-X on the AMD system for message sizes from 64K bytes to 1M bytes using OSU micro-benchmarks.



(a)



(b)



(c)

Figure 18: Performance comparison of HINT against Intel-MPI and MVAPICH2-X on the Intel Xeon Platinum system (Cascade Lake) for different problem sizes and PPNs in P3DFFT.

P3DFFT implements Fast Fourier Transforms (FFTs) in 3-dimensions by performing distributed matrix transpositions with MPI and domain decomposition. Every iteration of the algorithm contains a forward and a backward transform, with each transform calling an Alltoall with the receive buffer of the forward transform used as the send buffer in the backward transform. In this section, we evaluate the performance of our designs against state-of-the-art MPI libraries for various problem sizes using P3DFFT. We only show results on the Intel Cascade Lake system as we could not get HPC-X to build with P3DFFT on the other two systems, but the trends should remain the same.

Figure 18 shows the performance comparison of HINT against Intel-MPI and MVAPICH2-X for different problem sizes and process counts. At 8PPN, we see **4.8%** improvements over Intel-MPI and **3.7%** over MVAPICH2-X for 128x256x256 grids. The benefits increase for the 256x256x512 grid where we outperform Intel-MPI by **15.1%** and MVAPICH2-X by **12.2%**. We observe similar trends for 16PPN and 32PPN. At 16PPN, our proposed solution outperforms Intel-MPI by **11.2%** and MVAPICH2-X by **13.2%** for 256x256x512 grids. For 256x512x512, the benefits increase to **22.2%** over Intel-MPI and **21.9%** over MVAPICH2-X. Similarly, at 32PPN, our designs outperform Intel-MPI and MVAPICH2-X by up to **11%**. We observe improvements in a wide range of problem sizes and processes per node due to having hybrid memory copy ordering that outperforms state-of-the-art by a significant margin for small/medium messages. Problem sizes using larger messages benefit from the non-temporal memory copy scheme.

VII. Related Work

MPI optimizations and tuning are widely studied, involving solutions that range from improving communication patterns [16], [17], reducing latency [18], leveraging hardware architectures and network [19], enhancing scalability [20], increasing overlap between communication and computation [21], and minimizing memory usage [22]. Modern MPI libraries often tune between bruck, recursive doubling, scatter destination, and pairwise algorithms [3]. [16] proposes a rank re-ordering approach based on application characteristics. Our designs work on any ordering of ranks and can benefit from application-aware mapping. [23], [24] analyze and model the performance impact and limitations of cache coherence protocols on MPI communication. [25] proposes the detection and elimination of non-temporal memory accesses to reduce cache pollution on multicores for mixed workloads of independent applications. [26] provides a performance analysis of several MPI collective operations and models the time complexity of the linear and pairwise-exchange MPI_Alltoall algorithms.[27] proposes cache-oblivious MPI_Alltoall algorithms to allocate send and receive buffers on a shared heap using Morton order. Implementing a shared heap requires modification of the memory allocator, which interferes with custom allocator implementations in state-of-the-art MPI libraries. Our designs also work on processes with private heaps. The design adopted by [27] only uses Morton curves, but our design is generic and uses a combination of various memory copy orderings. The benefits of this approach are explained in Section V-D and shown in Figure 6. Moreover, our design addresses bandwidth bottlenecks for large messages using non-temporal instructions. [28] suggests a dynamic selection scheme of MPI_Alltoall algorithms based on the system and workload using the P-LogP model. Several other studies exploit the underlying network features such as Infiniband HCA Gather-Scatter [29], multi-HCA systems [30], and Smart Network Interface Cards (SmartNICs) such as NVIDIA's BlueField Data Processing Units (DPUs) [31], [32] to offload and improve the performance of MPI_Alltoall. The inter-node solutions in [31], [32], [29], [3], [16], [17] are orthogonal to our designs and can benefit from our intra-node optimizations.

VIII. Conclusion and Future Work

Dense collective patterns such as MPI_Alltoall have a significant number of memory transactions as every process has to send/receive a chunk of data to/from every other process. The ordering of memory copies during the MPI_Alltoall operation can have a large effect on performance. The best-performing access pattern depends on various factors, including cache coherence protocols, associativity, cache size, and other architecture-specific features. In this paper, we first propose three different orders for performing Alltoalls, including Hilbert-based curves, row-wise and column-wise orders. We analyze trends at different process counts and propose a hybrid Alltoall scheme using XPMEM that switches between orders depending on the message size and process count. We then find bottlenecks in memory transactions for large messages and propose the usage of non-temporal instructions to improve bus bandwidth. Our proposed solutions reduce the latency versus state-of-the-art libraries by up to **10x** for micro-benchmarks and up to **22.2%** for the P3DFFT application. In future work, we intend to explore extensions to our designs for non-x86 architectures such as ARM.

REFERENCES

[1] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.0*, Jun. 2021. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[2] AMD EPYC 4th generation 9004 processors, https://www.amd.com/en/products/processors/server/epyc/4th-generation-9004-and-8004-series.html.

[3] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of Collective Communication Operations in MPICH," *Int. J. High Perform. Comput. Appl.*, vol. 19, no. 1, p. 49–66, feb 2005. [Online]. Available: https://doi.org/10.1177/1094342005051521

[4] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-Oblivious Algorithms," *ACM Trans. Algorithms*, vol. 8, no. 1, jan 2012. [Online]. Available: https://doi.org/10.1145/2071379.2071383

[5] D. Hilbert and D. Hilbert, "Über die stetige Abbildung einer Linie auf ein Flächenstück," *Dritter Band: Analysis· Grundlagen der Mathematik· Physik Verschiedenes: Nebst Einer Lebensgeschichte*, pp. 1–2, 1935.

[6] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi, "Recursive array layouts and fast parallel matrix multiplication," in *Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, 1999, pp. 222–231.

[7] J. Peng, J. Fang, J. Liu, M. Xie, Y. Dai, B. Yang, S. Li, and Z. Wang, "Optimizing MPI Collectives on Shared Memory Multi-Cores," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3581784.3607074

[8] PAPI, https://icl.utk.edu/papi/.

[9] OSU Micro-benchmarks, http://mvapich.cse.ohio-state.edu/benchmarks/.

[10] J. McCalpin, "Memory bandwidth and machine balance in high performance computers," *IEEE Technical Committee on Computer Architecture Newsletter*, pp. 19–25, 12 1995.

[11] XPMEM, https://github.com/hpc/xpmem.

[12] D. Deford and A. Kalyanaraman, "Empirical Analysis of Space-Filling Curves for Scientific Computing Applications," in *2013 42nd International Conference on Parallel Processing*, 2013, pp. 170–179.

[13] P. Mckenney, "Memory Barriers: a Hardware View for Software Hackers," 08 2010.

[14] C. H. Hamilton and A. Rau-Chaplin, "Compact Hilbert indices: Space-filling curves for domains with unequal side lengths," *Information Processing Letters*, vol. 105, no. 5, pp. 155–163, 2008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020019007002153

[15] D. Pekurovsky, "P3DFFT: A Framework for Parallel Computations of Fourier Transforms in Three Dimensions," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C192–C209, 2012. [Online]. Available: https://doi.org/10.1137/11082748X

[16] G. Mercier and E. Jeannot, "Improving MPI Applications Performance on Multicore Clusters with Rank Reordering," in *Recent Advances in the Message Passing Interface*, Y. Cotronis, A. Danalis, D. S. Nikolopoulos, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 39–49.

[17] G. Chochia, D. Solt, and J. Hursey, "Applying on Node Aggregation Methods to MPI Alltoall Collectives: Matrix Block Aggregation Algorithm," in *Proceedings of the 29th European MPI Users' Group Meeting*, ser. EuroMPI/USA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 11–17. [Online]. Available: https://doi.org/10.1145/3555819.3555821

[18] P. M. Mattheakis and I. Papaefstathiou, "Significantly Reducing MPI Intercommunication Latency and Power Overhead in Both Embedded and HPC Systems," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, jan 2013. [Online]. Available: https://doi.org/10.1145/2400682.2400710

[19] B. Ramesh, K. K. Suresh, N. Sarkauskas, M. Bayatpour, J. M. Hashmi, H. Subramoni, and D. K. Panda, "Scalable MPI Collectives using SHARP: Large Scale Performance Evaluation on the TACC Frontera System," in *2020 Workshop on Exascale MPI (ExaMPI)*, 2020, pp. 11–20.

[20] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "PaRSEC: Exploiting Heterogeneity to Enhance Scalability," *Computing in Science and Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[21] V. Marjanović, J. Labarta, E. Ayguadé, and M. Valero, "Overlapping Communication and Computation by Using a Hybrid MPI/SMPSs Approach," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 5–16. [Online]. Available: https://doi.org/10.1145/1810085.1810091

[22] S. Sur, M. J. Koop, and D. K. Panda, "High-Performance and Scalable MPI over InfiniBand with Reduced Memory Usage: An in-Depth Performance Analysis," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 105–es. [Online]. Available: https://doi.org/10.1145/1188455.1188565

[23] B. Putigny, B. Ruelle, and B. Goglin, "Analysis of MPI Shared-Memory Communication Performance from a Cache Coherence Perspective," in *2014 IEEE International Parallel and Distributed Processing Symposium Workshops*, 2014, pp. 1238–1247.

[24] G. Chehaibar, M. Zidouni, and R. Mateescu, "Modeling Multiprocessor Cache Protocol Impact on MPI Performance," in *2009 International Conference on Advanced Information Networking and Applications Workshops*, 2009, pp. 1073–1078.

[25] A. Sandberg, D. Eklöv, and E. Hagersten, "Reducing Cache Pollution Through Detection and Elimination of Non-Temporal Memory Accesses," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010, pp. 1–11.

[26] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. Fagg, E. Gabriel, and J. Dongarra, "Performance analysis of MPI collective operations," in *19th IEEE International Parallel and Distributed Processing Symposium*, 2005, pp. 8 pp.–.

[27] S. Li, Y. Zhang, and T. Hoefler, "Cache-Oblivious MPI All-to-All Communications Based on Morton Order," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 542–555, 2018.

[28] H. N. Mamadou, T. Nanri, and K. Murakami, "A robust dynamic optimization for MPI Alltoall operation," in *2009 IEEE International Symposium on Parallel and Distributed Processing*, 2009, pp. 1–15.

[29] A. Gainaru, R. L. Graham, A. Polyakov, and G. Shainer, "Using InfiniBand Hardware Gather-Scatter Capabilities to Optimize MPI All-to-All," in *Proceedings of the 23rd European MPI Users' Group Meeting*, ser. EuroMPI '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 167–179. [Online]. Available: https://doi.org/10.1145/2966884.2966918

[30] K. K. Suresh, A. P. Guptha, B. Michalowicz, B. Ramesh, M. Abduljabbar, A. Shafi, H. Subramoni, and D. Panda, "Efficient Personalized and Non-Personalized Alltoall Communication for Modern Multi-HCA GPU-Based Clusters," in *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2022, pp. 100–104.

[31] K. K. Suresh, B. Michalowicz, B. Ramesh, N. Contini, J. Yao, S. Xu, A. Shafi, H. Subramoni, and D. Panda, "A Novel Framework for Efficient Offloading of Communication Operations to Bluefield SmartNICs," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023, pp. 123–133.

[32] M. Bayatpour, N. Sarkauskas, H. Subramoni, J. Maqbool Hashmi, and D. K. Panda, "BluesMPI: Efficient MPI Non-Blocking Alltoall Offloading Designs on Modern BlueField Smart NICs." Berlin, Heidelberg: Springer-Verlag, 2021, p. 18–37. [Online]. Available: https://doi.org/10.1007/978-3-030-78713-4_2